Model checking a TTCAN implementation

Daniel Keating, Allan McInnes and Michael Hayes
University of Canterbury
Electrical and Computer Engineering
Christchurch, New Zealand
daniel.keating@pg.canterbury.ac.nz, allan.mcinnes@canterbury.ac.nz, michael.hayes@canterbury.ac.nz

Abstract

This paper describes how the SPIN model checker has been applied to find and correct problems in the software design of a distributed vessel control system currently under development at a control systems specialist in New Zealand. The system under development is a mission critical control system used on large marine vessels. Hence, the requirement to study the architecture and verify the implementation of the system. The model checking work reported here focused on analysing the implementation of the Time-Triggered Controller-Area-Network (TTCAN) protocol, as this is used as the backbone for communications between devices and thus is a crucial part of the control system. The starting point was to develop a set of general techniques for model checking TTCAN-like protocols. The techniques developed include modelling the progression of time efficiently in SPIN, TTCAN message transmission, TTCAN error handling, and CAN bus arbitration. These techniques form the basis of a set of models developed to check the implementation of TTCAN in the control system as well as the fault tolerance schemes added to the system. Descriptions of the models and properties developed to check the correctness of the implementation are given, and verification results are presented and discussed. This application of model checking to an industrial design problem has uncovered and corrected a number of potentially costly issues in the original design.

1. Introduction

A New Zealand based control systems specialist is currently developing a mission critical control system to be used on large marine vessels. The Time-Triggered Controller-Area-Network (TTCAN) protocol is used for communication between modules on the system. In distributed systems such as this, where there is communication between concurrently executing processes, the complicated nature of the systems often leads to design errors difficult to detect using conventional testing methods. The aim of this research was to apply model checking techniques to assist in verifying the correct operation of the implementation of the TTCAN protocol in the control system.

Section 2 gives an overview of model checking, the TTCAN protocol, and the implementation of the control system. Section 3 describes the set of techniques developed for model checking TTCAN-like protocols. Section 4 describes the model developed to verify the TTCAN implementation. Section 5 describes the correctness properties developed to check against the model and presents the verification results. Finally, Section 6 concludes the paper.

2. Background

This section briefly introduces model checking, gives an overview of the TTCAN protocol, and describes how it is implemented in the control system.

2.1. Model checking

Model checking is a technique used to automatically verify correctness properties against a finite state model of a software or hardware system. A concurrent program can be visualised as a Finite State Machine (FSM) or automaton. The state of the system is the current expression being executed in each of the concurrent processes and the corresponding set of values of variables. The FSM contains nodes to represent every possible state the system enters and edges to represent possible transitions between states [3]. A model-checker traverses all possible paths through the concurrent system's FSM, checking specified safety and liveness properties at each step. If a property is disproved a counter-example is generated showing the sequence of events leading to the violation of the property. The counter-example is valuable for debugging a system [2].

One model-checker that has become popular in industry is the SPIN model checker. The input specification

language to SPIN for describing a model is called Process Meta-Language (PROMELA). Correctness properties to be checked against the model are described using Linear Temporal Logic (LTL). LTL formulae are logical statements that describe sequences of a program's states [1]. A process is created in PROMELA to represent the control flow of a program. The individual threads of a multi-threaded application, or simultaneously executing nodes on a distributed network, are modelled in PROMELA by creating multiple processes. Shared variables and message channels are used to model interprocess communication.

2.2. TTCAN protocol

Modern vehicles contain complex distributed control systems. Currently, the CAN protocol is one of the most popular communications protocols used in these types of networks [6]. CAN is an asynchronous event-triggered protocol, meaning that events are sent around the network as they occur. Due to the event-triggered nature of the protocol, conflicts can occur on the bus when multiple messages are sent simultaneously. A priority-based arbitration scheme determines which message is transmitted on the bus. This introduces non-deterministic latency to message transmissions, complicating the design of systems with tight timing constraints, such as the closed-loop distributed control system used in vehicle brake or steer by-wire systems [8].

The increasing size and complexity of these vehicle control systems has introduced a need for a variant of the CAN protocol that provides deterministic timing across the system. This simplifies system design and enables design of more complicated systems. Time-Triggered CAN (TTCAN) is a variant of the CAN protocol that offers more precise timing.

The TTCAN protocol adds a session layer (layer 5 of the OSI model), defined by ISO 11898-4, to the existing data link (OSI layer 2) and physical (OSI layer 1) layers of the existing CAN protocol [7]. It is a synchronous (time-triggered) protocol, where the transmission of messages is based on the progression of a globally synchronised time base. Each node has a pre-defined schedule of messages to transmit in pre-allocated time-slots to eliminate bus conflicts and guarantee message latencies [8].

Time synchronisation between TTCAN nodes is achieved by the periodic transmission of a specific message known as a 'reference message' from a designated time-master node. On receiving the Start of Frame (SOF) bit of the reference message, all node transmission cycles are restarted. Messages are then sent when their scheduled time-slot becomes active. The period elapsed between two consecutive reference messages is known as a 'basic-cycle', and a number of 'basic-cycles' with different message schedules may be repeated in a 'matrix-cycle' [6].

Using a time-triggered protocol allows deterministic timing of transmissions and a higher portion of the overall bandwidth of the system to be utilised. However, the latency of transmissions may be greater than when using an event-triggered protocol.

2.3. A TTCAN implementation

The system to be analysed is a dual redundant distributed vessel control system based on the TTCAN protocol. The redundant CAN buses are labelled `P` and `Q` in Figure 1. The vessel is usually wired with a bus running down each side of the vessel. The system consists of three main types of modules: Control Input Devices (CID), Master Controllers (MC), and Hydraulic Controllers (HC). Each module is connected to both of the redundant CAN buses.

CIDs take input actions from the operator such as adjusting the throttle or steering the vessel and translate these to input commands that are sent to the MC. The CIDs have redundant microcontrollers, each connected to one of the redundant CAN buses.

The MC processes control input commands from the CIDs and feedback messages from the HCs. It translates these commands and feedback messages into output demands that are sent to the HC. The microcontroller on the MC has two CAN controller modules, one connected to the `P` bus and the other to the `Q` bus, as shown in Figure 1. When the currently active time-master MC transmits messages, they are sent on both CAN buses. The MC is the time-master of the system; it is responsible for synchronising the message schedules of the other network nodes. This is achieved by the active MC periodically sending a TTCAN sync reference message.

3. Modelling TTCAN protocols

Our approach to modelling TTCAN is influenced by techniques developed by Weininger and Cofer when modelling the ASCB-D synchronisation algorithm with SPIN. We use an abstract simulation of message transmission and time progression to constrain the possible interleavings of events and reduce the state-space required for verification. We also combined handling message transmission and time progression into a single process to further reduce the resources required for verification [5].

As the focus of this work is verification of the TTCAN protocol, it is not necessary to model the underlying CAN protocol in detail. Our model includes only the properties of the CAN protocol that are relevant for verification of the TTCAN layer, such as arbitration and bus access. Using this abstracted approach not only reduces the resources re-

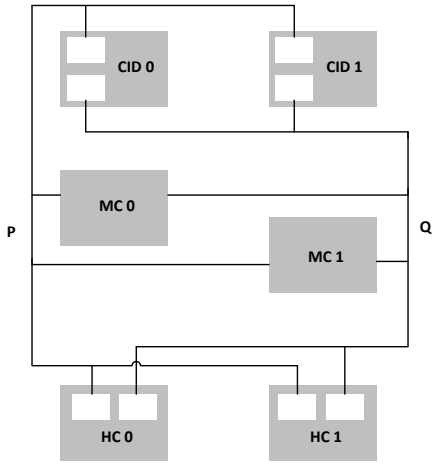

Figure 1. Block diagram of the redundant vessel control system.

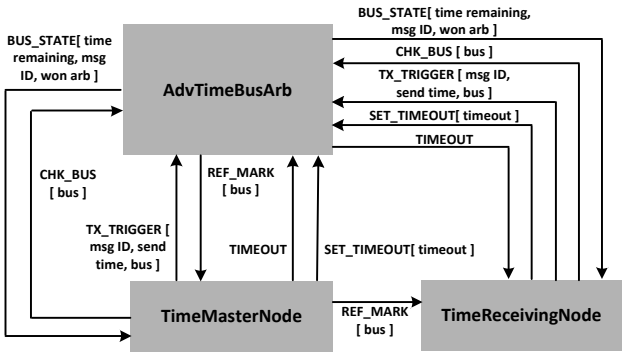

Figure 2. Block diagram showing processes and sync channel messages for handling timing and message transmission used in the control system model.

quired for verification but also makes the model simpler to understand and extend.

3.1 Modelling time progression in TTCAN protocols

The strategy used to model the progression of time is commonly known as Variable Time Advance (VTA). Using this strategy, the current time advances by having the model jump to the instant where the next event causes a state transition. The advantage of using this technique over a fixed-time step model is that periods where there is no activity are skipped, allowing the state-space of the verification to be reduced. An untimed model was considered, but a timed strategy was used to allow modelling of timing of an event relative to another [5]. We found that using a timed approach simplified modelling of the ordering of events at different nodes.

Both the progression of time and CAN bus arbitration in the model is handled by the `AdvTimeBusArb` process. The interaction between processes, through sync channels, to model the progression of time is shown in Figure 2. The sequence of events that advance time to the next scheduled event that causes a state transition is shown by the flow chart in Figure 3 and the message sequence chart in Figure 4. The sequence of events for handling the progression of time and bus arbitration in `AdvTimeBusArb` are:

1. The process compares each node's `nextTimeout` value, the time until the node's next scheduled event, to find the node with the minimum timeout value.

2. Bus arbitration is handled if time has advanced and there are any messages triggered for transmission.

3. A `TIMEOUT` sync channel message is sent to the node that is next to timeout. This timeout indicates an event is due to be triggered at the node. If multiple nodes are triggered simultaneously, they are looked at each in turn with the lowest identifier triggered first. As simultaneously transmitted messages are buffered, then arbitration decided before time is next advanced, this order is not important.

4. `AdvTimeBusArb` waits for `CHK_BUS`, `TX_TRIGGER`, or `REF_MARK` sync channel messages. On receiving `CHK_BUS`, the current state of the bus is returned to the requesting node process. `TX_TRIGGER` initialises transmission of a message on the bus. `REF_MARK` indicates a reference message has been received at a node and each node's next timeout value is to be updated.

5. `AdvTimeBusArb` waits for a `SET_TIMEOUT` sync channel message from the currently executing node. An updated timeout value is sent from the node as a sync channel parameter, and this is used to update the node's next timeout value.

6. The time elapsed by the node that timed-out is subtracted from the other node's next timeout values.

7. `AdvTimeBusArb` then restarts to find the next node with the minimum time to next timeout.

3.2. Modelling CAN bus arbitration

Listing 1 shows how bus arbitration of simultaneously transmitting nodes is resolved. When a message is initially triggered for transmission on the bus

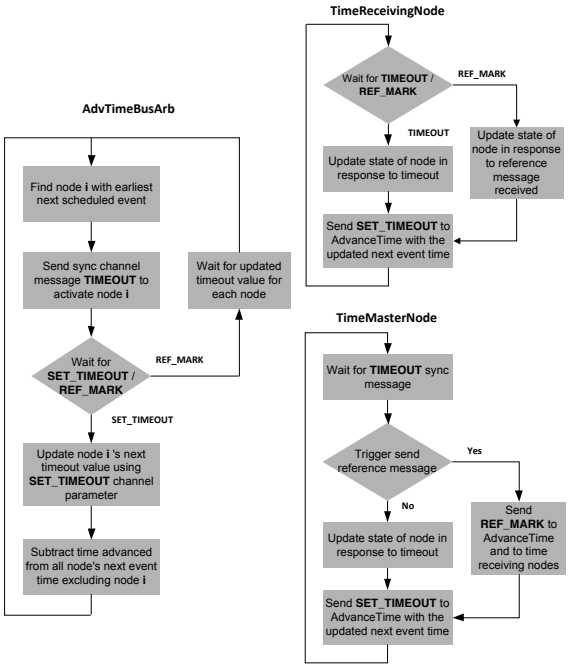

Figure 3. Sequence of events for advancing time in the "AdvTimeBusArb", time-master node, and time-receiving node processes.

the `doArbitration` flag within the `AdvTimeBusArb` process is set. The `AdvTimeBusArb` process stores the identifier of any node transmitting at this instant in the `AdvTimeBusArb` processes' `nodeFrameId` array. Once time advances, arbitration is handled with the node transmitting the highest priority message winning (transmitting the lowest identifier). The identifier of the winning node's message is assigned to `BusFrameId`, which is used to keep track of the current state of the bus.

3.3. Handling TTCAN reference messages

In this TTCAN protocol model, reference messages are able to cause state-transitions that affect the next timeout value in a receiving node. They may be received before the next scheduled timeout at a node and alter the timeout depending on the current state of the node. This is accounted in the model by sending the `REF_MARK` sync channel message between the transmitting and receiving processes when a reference message is transmitted, as shown in Figure 2. A timeout in the receiving node may be interrupted and updated to a new value on receiving `REF_MARK`.

Listing 2 gives an example of how handling the reference message has been implemented in the model. If a `REF_MARK`, due to the node receiving a reference message, is received from another node during the period after

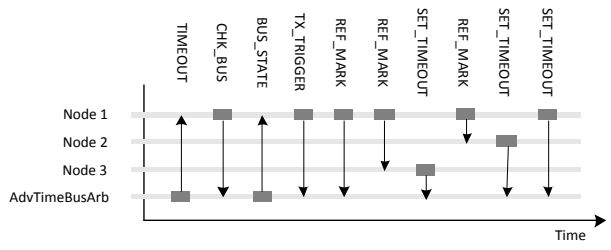

Figure 4. Message sequence chart showing the sequence of sync channel messages sent when transmitting a reference message.

Listing 1. PROMELA source from "AdvTimeBusArb" process for bus arbitration.

```
IF ((busArb[checkBus].doArbitration != false)
    && (minNextTimeout > 0)) ->
    busArb[checkBus].doArbitration = false;
    busArb[checkBus].tempBusId = BUS_IDLE;
    /* Find the highest priority message of
    currently transmitting nodes. */
    FOR(i, 0, TOTAL_NODES)
        IF busArb[checkBus].nodeFrameId[i]
            < busArb[checkBus].tempBusId ->
            busArb[checkBus].tempBusId =
                busArb[checkBus].nodeFrameId[i];
            busArb[checkBus].arbWinnerIndex = i;
        FI;
    ROF(i);
    busArb[checkBus].busFrameId.messageID
        = busArb[checkBus].tempBusId;
FI;
```

the initial reference message is transmitted from a potential time-master node, then the process updates the next timeout value to `REF_TRIGGER + refTriggerOffset`. On transmitting a reference message, a `REF_MARK` sync channel message is also sent from the transmitting process to `AdvTimeBusArb`. This causes `AdvTimeBusArb` to change to a state where it allows nodes that received the reference message to transmit an updated next timeout value through `SET_TIMEOUT` if necessary.

4. Model checking the implementation of TTCAN

This section describes the models developed for model checking the vessel control system. The model checking work here focuses on the implementation of the TTCAN protocol, as this is a major component of the control system. The approach taken was to develop multiple models focused on specific aspects of the system, rather than one monolithic

Listing 2. PROMELA source from "TimeMasterNode" for handling received reference message during a timeout after transmitting the first reference message.

```
if
:: TIMEOUT[nodeNum] ? 0 ->
    /* Scheduled timeout has elapsed. */
:: REF_MARK[nodeNum] ? 0 ->
    /* Received reference message. */
    CHK_BUS[nodeNum] ! bus;
    BUS_STATE[nodeNum] ? _, busFrameIdTemp, _;
    /* Check received ref message priority
    against the node's priority. */
    if
    :: busFrameIdTemp < refMsgID ->
        SET_TIMEOUT[nodeNum] ! (REF_TRIGGER
            + refTriggerOffset);
        goto TIMEOUT_4;
    :: busFrameIdTemp > refMsgID ->
        refTriggerOffset = 0;
        SET_TIMEOUT[nodeNum] ! (REF_TRIGGER
            + refTriggerOffset);
        goto TIMEOUT_3;
    fi;
fi;
```

model. It was found that developing a number of separate smaller models was essential in checking this system due to its size and complexity. Using this technique, we were able to model the system efficiently without encountering the state space explosion problem. Three PROMELA models of different parts of the implementation are described, along with sets of correctness properties for each model. The properties have been verified against the models using the SPIN model checker. The models described are:

- A model of the implementation of TTCAN. This is used to check the basic operation of the system with the minimum, most common, and maximum number of modules that can be configured for the system.
- A model of the active time-master election process, called the "voter" process in the implementation, analyses the election procedure that determines the active time-master on startup or reintegration.
- A model of the module in the MC that selects which of the redundant buses the active MC listens to, called the "signal picker" in the implementation.

After talking with the development engineers and analysing the code, it appeared that the correct operation of the redundant MC nodes is crucial to the safety of their system, and checking the correctness of the "voter" procedure is essential to the correct operation of the MC. The periodic sync reference message sent from the currently active time-master MC is the heart-beat of the system. The reference message triggers and synchronises the transmission schedules of all the other nodes in the system. Without the sync reference message, exclusive window messages, responsible for control of the vessel, are not transferred from the helm and throttle to the hydraulic controller. The TTCAN protocol model also allowed us to verify the correct transfer of the scheduled exclusive window messages. Also, crucial to the correct operation of the MC is the "signal picker" module. This is responsible for selecting which of the redundant buses the MC is currently listening to.

4.1. TTCAN protocol model

The TTCAN protocol model is used to check the implementation of the TTCAN protocol in the control system. The model focuses on verifying the correct transmission of the periodic sync reference messages and the scheduled exclusive window messages, due to their importance to the correct operation of the system.

The TTCAN protocol model is constructed using the techniques for modelling TTCAN-like protocols described in Section 3. The TTCAN protocol implementation, and our model of it, differs from ISO TTCAN in several ways:

- Exclusive window messages can overrun into the next time-slot causing a delay in the following scheduled message.
- There are multiple TTCAN transmit buffers.
- There is no ISO TTCAN error handler and startup synchronisation scheme present in the implementation.

In the TTCAN protocol model MC nodes are the potential time-master nodes in the system; each MC is modelled by a `TimeMasterNode` process. HC and CID nodes are time-receiving nodes; these are modelled by `TimeReceivingNode` processes. CAN bus arbitration and handling of timing are combined in a process called `AdvTimeBusArb`, as shown in Figure 2.

The model can be conditionally compiled to check three different system configurations: a minimal configuration (2 MC nodes, 1 HC, and 1 CID), the most common setup (2 MCs, 2 HCs, and 2 CIDs), and a configuration with the maximum number of nodes (2 MCs, 4 HCs, and 5 CIDs).

4.2. "Voter" model

The "voter" model is used to verify the active time-master election process used in the implementation to select an active and backup time-master during startup of a MC node. Figure 6 shows the "voter" state machine that is the

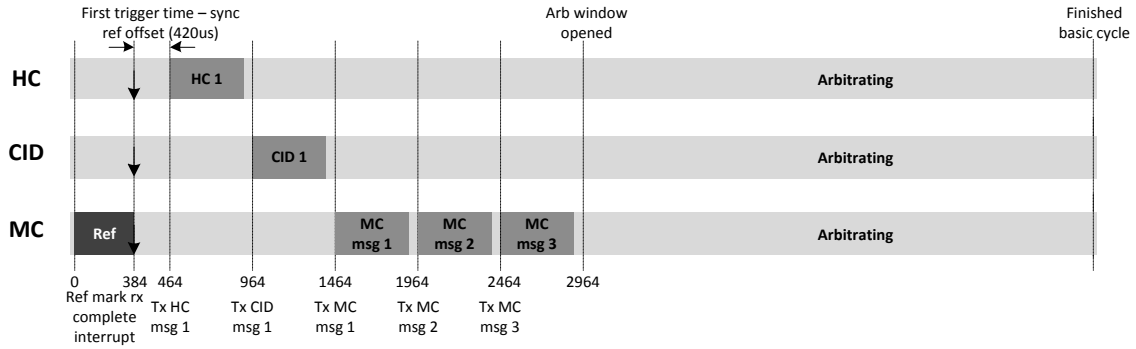

Figure 5. Timing diagram for 2 MC, CID, and HC model transmitting 480 μs messages.

basis for the model of the active time-master election procedure. Scenarios where intermittent faults cause time-master nodes to periodically toggle on and off are modelled, as well as the addition of initial startup delays of the MC nodes.

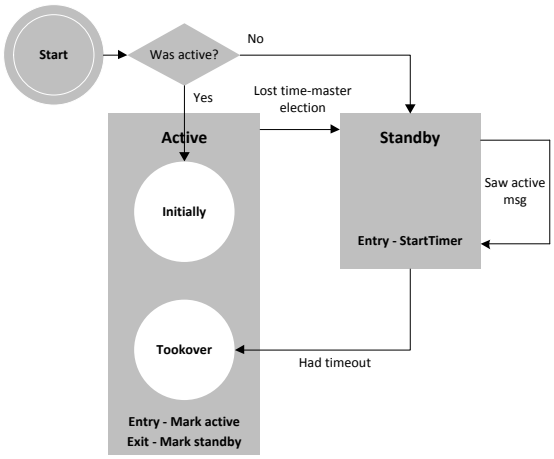

Figure 6. State machine of time-master election process in MC nodes.

The "voter" model consists of two MC nodes, a CID, and a HC. The MC nodes are represented in the model by `TimeMasterNode` processes and each is associated with a `PeriodicTimer` process. In this model, the MCs simulate the "voter" procedure. The `AdvTimeBusArb` process is responsible for handling the progression of time, message transmissions, and CAN bus arbitration in the model.

4.3. "Signal picker" model

The "signal picker" is a module that exists in the MC and is responsible for selecting which of the redundant buses the MC is currently listening to. The "signal picker" model verifies the module's interaction with the environment. The model simulates messages received on either bus, messages reporting errors from their source, dropped messages, and the periodic update of the state of the module as occurs in the implementation. Correctness properties have been developed based on the developer's design specifications. Assertions are used in the model to check the properties hold. The model checks each of the possible interleavings of events between the signal picker and the redundant buses. In this case, an untimed model has been used; it was not necessary to include the timing of events relative to each other to check the properties specified.

The model consists of a process that represents the "signal picker" module, and a process, called `TriggerInputs`, that models the interaction of the "signal picker" module with its environment. The "signal picker" reacts to events triggered by the `TriggerInputs` process.

The `TriggerInputs` process non-deterministically triggers events that occur in the environment and in the "signal picker", such as: receiving messages, simulating missed (dropped) messages, and initialising the periodic update of the state of the "signal picker". The periodic update is triggered every 1.5 s in the implementation, but is triggered non-deterministically in the model as the update could occur at any time relative to the messages being received on

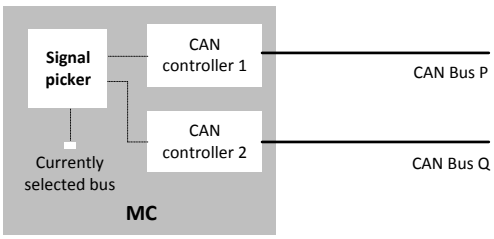

Figure 7. Diagram showing "signal picker" module's connections within the MC.

the bus.

The "signal picker" process model makes decisions about whether or not to swap buses whenever a new message is received. Received messages are non-deterministically determined to be either messages received without error or messages indicating sensor errors at the message source. The model also includes a check that the rules related to swapping to the redundant bus are not broken after a new message has been received or the state of the module has been updated.

5. Verification of models

Each model can be conditionally compiled to check a number of different scenarios that potentially cause failures in the system. Sets of correctness properties have been developed, based on the developer's design specifications, to check against the models. Verification of the models helps to gain confidence that the implementation meets the original design specifications.

5.1 TTCAN protocol model

The correctness properties chosen to verify the TTCAN protocol model focus on checking the correct transmission of the scheduled exclusive window messages. This is due to their importance to the correct operation of the system, as exclusive window messages are responsible for transferring control information from the user inputs to the hydraulic control units. Also, as they are transmitted in one-shot mode (without CAN retransmission), it is essential that they are either transmitted correctly or errors detected and reported if this is not possible. To check the correctness of the exclusive window messages, safety properties are used to ensure no scheduled messages fail to be received at a node and no unexpected messages are received at a node. The correctness properties have been verified against the model under different potential failure scenarios.

The failure scenarios modelled are as follows:

1. The startup delay of the MC nodes is staggered by non-deterministically choosing an initial startup delay.
2. Both MCs are initialised as active time-masters.
3. The states of each time-master node are swapped from active to backup and backup to active after the initial basic-cycle completes.
4. The first HC message is delayed to simulate an accumulated overrun when a number of nodes transmit messages longer than their allocated time-slots.
5. Delay of the first CID node's exclusive message.
6. Delay of the first MC node's exclusive message.
7. The backup MC is configured as a "babbling idiot" node. The node repeats sending of an unexpected message through the system's exclusive time-slots, ignoring the scheduled message trigger times.
8. The backup MC is configured with a simulated configuration error. The node sends unexpected messages at the beginning of each scheduled exclusive time-slot.
9. Reference messages sent by the active time-master are not received correctly at other nodes, but are received correctly at the time-master node (meaning there is no retransmission).
10. Reference messages sent by the active time-master are not received correctly at all nodes including the time-master node causing a timeout and retransmission.

The following list of 6 correctness properties have been verified against the model with the "base" configuration:

1. Absence of violated assertions and deadlock in model.
2. $\Box$ `!(n1_rx_error || n2_rx_error || n3_rx_error || n4_rx_error)` — Absence of scheduled message receive error at nodes 1 – 4.
3. $\Box$ `!(n1_schedule_err || n2_schedule_err || n3_schedule_err || n4_schedule_err )` — Absence of received message that has not been scheduled at nodes 1 – 4.
4. $\Box \Diamond$ `n1_alive` — The active time-master node (`Node 1`) always eventually starts a new basic-cycle.
5. $\Box \Diamond$ `n2_n4_recv_sync_ref` — Nodes 2 – 4 always eventually receive a reference message triggering the restart of a new basic-cycle.
6. $\Box \Diamond$ `n1_n4_arb_window_open` — The arbitration window in nodes 1 – 4 is always eventually open.

The correctness properties have been verified against the three configurations of the TTCAN model described in Section 4.1. For each property verified against the models, the models are configured with the failure scenarios described described above or a default configuration. In the default configuration, both MC nodes start simultaneously. One MC is configured as the active time-master; the other is configured as the backup. The specified LTL formulae, scenarios modelled, and the verification results of each property checked against the "base" configured model with both 480 μs and 636 μs messages are summarised in Table 1. The first correctness property, checking for an absence of deadlocks and assertions, passed verification without error when checked against each of the failure scenarios

modelled. Figure 5 shows a timing diagram for messages 480 μs in length transmitted on the CAN bus in a system setup with the "base" configuration.

Table 1. Verification results for "base" configured model with 480 μs and 636 μs messages.

Property	Failure scenario	Correctness property	Errors (480 μs)	Errors (636 μs)
1	Default	2	0	0
2	1	2	0	0
3	2	2	0	0
4	3	2	0	0
5	4	2	0	1
6	5	2	0	1
7	6	2	0	1
8	7	2	0	0
9	8	2	1	1
10	Default	3	0	0
11	1	3	0	0
12	2	3	0	0
13	3	3	0	0
14	7	3	0	0
15	Default	5	0	0
16	Default	6	0	0
17	Default	7	0	0
18	9	7	0	0

In verification of the "base" model using 480 μs messages:

- 17 of 18 properties checked passed verification.
- Property 9 fails as a message scheduled to be received is lost when an unconfigured node is added.

Using 636 μs messages:

- 14 of 18 properties checked passed verification.
- Property 9 fails verification.
- Property 5 fails verification. When an extra delay of 450 μs is added before transmitting the HC's scheduled message, the first message to be sent from the MC is lost and a verification error is reported. The delay causes the HC's message to be transmitted over the CID's and MC's exclusive time-slots. Once the HC's message completes, both the pending messages will be sent simultaneously. CAN arbitration will cause the lower priority MC message to be lost, causing the verification error to be reported. The delay has been artificially added to the model and is of arbitrary length, but this could occur in a system that has a larger configuration, as the delay due to each exclusive message sent over-running into the next time-slot will accumulate. The error occurs when an exclusive window message transmits over two other exclusive window message triggers. The exclusive messages from both nodes are delayed until the currently transmitting node has completed. When the current message finishes both nodes will send their messages simultaneously and one of the pending messages will be lost due to CAN bus arbitration. As the exclusive messages are sent in one-shot mode, the lost message will not be retransmitted.
- Properties 6 and 7 fail verification. Delaying the CID and first MC messages (delayed by 450 μs), cause verification errors. In both cases, the last message sent by the MC is lost. In the CID case, this is due to the CID message transmitting over the first and second scheduled MC message trigger points. When the third MC message is triggered the node's backup buffer is already full and the message is dropped. In the MC case, the first MC message is transmitting overtop of the trigger points for the second and third MC messages. The second message is put into the node's backup buffer but the third is dropped as the node is currently transmitting and the backup buffer is full.

Most importantly, in the process of creating the models, it was also found that in the implementation it is possible that exclusive window messages are lost without an error being passed to the application. Reporting an error is important in this case as these exclusive window messages are transmitted in one-shot mode without the usual CAN retransmission. An intermittent fault on a bus may cause a message to be repeatedly lost, causing the MC to switch to the redundant bus without reporting the fault. A possible solution to this problem is to add the ISO TTCAN error handling state machine to detect these errors by transitioning to an error state and signalling this to the application [7].

5.2 "Voter" model

The LTL correctness properties checked against the model to gain confidence that the implementation of the "voter" procedure meets the designer's specifications are:

1. $\Diamond\Box$ `(tm1_active && !tm2_active)` — The potential time-master node (`Node 1`) eventually always becomes the active time-master and `Node 2` becomes the backup time-master.
2. $\Diamond\Box$ `(tm2_active)` — `Node 2` eventually always becomes the active time-master if `Node 1` is disabled.

3. □◇ !(tm1_active && tm2_active) — Always eventually Node 1 and Node 2 are not both the active time-master.

4. ◇□ !(tm1_active && tm2_active) — Eventually always Node 1 and Node 2 are not both the active time-master.

5. □ !(tm1_active && tm2_active) — Always potential time-master nodes Node 1 and Node 2 are not both the active time-master.

6. □◇ (tm2_active) — Always eventually time-master node Node 2 is active when Node 1 is disabled following completion of the first basic-cycle.

7. Absence of violated assertions when the active time-master node (Node 1) is toggled on/off every 1.5 s.

8. Absence of violated assertions when the active time-master node (Node 1) is toggled on/off, and an initial bootup delay of the faulty active MC is added.

9. ◇□ (tm2_active) — Node 2 eventually always becomes the active time-master if there is a periodic fault causing the currently active time-master node (Node 1) to be toggled on and off every 1.5 s.

The results of the verification of the correctness properties checked against the "voter" model are:

- Properties 1 – 4 pass verification without error.
- Property 5 fails verification. The error trails reveal two situations where both potential time-masters are simultaneously configured as the active time-master. On the initial startup of the system, both time-masters become the active time-master after an initial timeout, the "voter" state-machine then determines the winning time-master. Also, if an active time-master is powered down the state is stored in non-volatile memory and when the node is reactivated, if the backup time-master has become active, both nodes will be in this state.
- Property 6 passes verification with the backup time-master taking over as expected.
- Property 7 fails when the on/off period is 3 s (1.5 s on and 1.5 s off) and passes for all other intervals tested. This kind of fault could potentially be caused by a faulty alternator sending a voltage spike to the MC's power supply, causing it to periodically switch on and off. The reason this period fails is because with the 3 s period the node is active just long enough so that the active time-master status message is able to be sent after startup, as illustrated in Figure 8. From tests of the MC hardware it was found the time-master status message is sent approximately 1.3 s after the node is powered up. This means, even with the 1.5 s periodic fault, the backup MC will think there is still an active MC on the network so will not timeout and takeover. In this case, 50% of the MC messages are effectively lost from the active MC due to the fault and the redundant MC will not takeover. An up down counter is used as a check to see if 1 or more messages have failed to be received at a node within a basic-cycle period. If this count becomes greater than 20 an assertion is triggered. This is the case in the model when the 3 s periodic fault is tested.

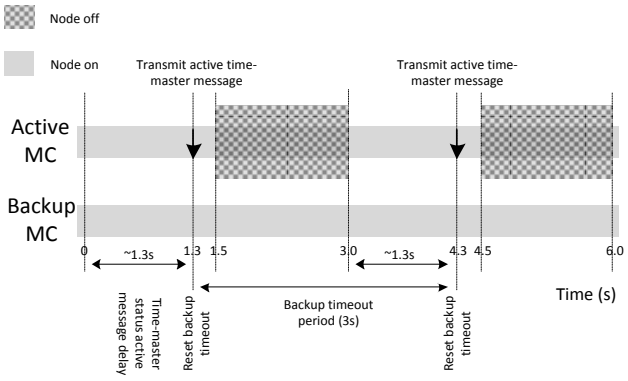

Figure 8. Timing diagram showing the sequence of events that causes the backup time-master to fail to takeover as the active time-master as expected.

- Property 8 passes when a delay is added to the MC.
- Property 9 fails verification showing that when there is 1.5 s periodic fault causing the active MC to toggle on and off the backup time-master MC does not take-over.

5.3 "Signal picker" model

The following properties checked are based on the developer's original design specification:

1. The module will only ever swap to the other bus if the current bus has received a message with an error value or a message has not been received since the status flags have been last reset. At least one message must be seen and no error messages may be received on the bus that is swapped to. An assertion is used to check this property after the model swaps buses.

2. If a message is dropped since the last time the modules flags are reset a transition is made to the redundant bus.

Property 1 passes verification; however, property 2 fails. Messages can be dropped on the bus the module is currently listening to without swapping to the backup bus. The "signal picker" only swaps to the redundant bus if no message has been seen (received correctly) on the current bus or a message is seen with an error flag set. This means if there is an intermittent fault on the bus it is possible to lose a number of message since the last periodic reset of the modules flags. In this case, as long as one message is received the module will not switch to the redundant bus.

6. Conclusion

We have applied model-checking techniques to a TTCAN-based vessel control system that is currently under development, and in the process learned several useful lessons about applying model-checking to industrial design problems. We found that the process of analysing and developing the finite state models of the implementation allows a developer to gain a deeper understanding of how the modules on a distributed system interact. As a result, a number of problems were found during the development of the models, before the verification phase of the model checking work. As expected, the majority of the problems identified tended to be in the interaction between modules across the distributed system. We also found that the creation and simulation of the models revealed sequences of events that were not taken into account during the initial software design. These events did not necessarily cause problems but this added to the understanding of some areas of the system. Adding this detail to the model ensures that these sequences of events are accounted for and will interact correctly with future additions to the system. Our experience has been that being on-site during development of the models was essential for communication between the modellers and the system developers. Talking with the developers helped to ensure that the model correctly represented the implementation, and that the potential problems that we found could be discussed and resolved.

By developing detailed models and correctness properties for the system early-on, areas were revealed where the initial design specifications had to be made more concise or altered. In this case, the specification of the backup MC takeover scheme, the length of the exclusive TTCAN message windows, and TTCAN error handling were all altered during the initial development. By identifying the problems early in development, the specifications could be updated and code modified with a minimum impact on project deadlines.

We found that developing a set of smaller models each targeted at different areas of the system allowed verification to be completed using a reasonable amount of resources. Due to the modular design of the system and low coupling between modules, developing a number of separate models was a logical step. We also found that using abstraction was key in developing models that could be verified using a reasonable amount of resources. During development, a number of modifications were made to the modelling of CAN message transmission and bus arbitration. Initially, a detailed layered model of the CAN protocol was developed, but to reduce the state-space of the verification an abstract model was used only including properties relevant to verification of TTCAN, such as bus access and arbitration. This freed up resources for verification of more complex network configurations and more properties relevant to this particular TTCAN implementation.

We found efficiently modelling the progression of time to be one of the most important aspects in the design of the models. Using the VTA technique proved to simplify the process of developing a model that accurately represents the protocol, when compared to an untimed model. We found that using VTA enabled greater control in constraining the sequences of events generated by the models, and this was important when modelling timing dependent areas of TTCAN such as the message schedules and the startup algorithm.

References

[1] Holzmann, G.J., The SPIN Model checker: primer and reference manual, Addison-Wesley, 2004.

[2] Stephan Merz. F. Cassez et al. (eds): Modeling and Verification of Parallel Processes, Springer-Verlag, LNCS 2067, pp. 3-38, 2001.

[3] Ben-Ari, M. Principles of the Spin Model Checker, Springer London, 2008.

[4] G. Leen and D. Heffernan, "Formal verification of the TTCAN protocol", 2002.

[5] Nicholas Weininger and Darren Cofer, "Modeling the ASCB-D Synchronisation Algorithm with SPIN: A Case Study", In Proceedings of the 7th SPIN Workshop, LNCS 1885, Springer-Verlag, 2000.

[6] Leen, G. and Heffernan, D., TTCAN: a new time-triggered controller area network, Microprocessors and Microsystems Journal, vol. 26, pp. 77-94, 2002.

[7] ISO 11898, Road vehicles Controller area network (CAN) Part 1 and Part 4, International Standards Organisation.

[8] Fuhrer, T., Muller, B., Dieterle, W., Hartwich, F., Hugel, R. and Walther, M., "Time triggered communication on CAN", Seventh International CAN Conference (ICC), Amsterdam, Netherlands, 2000.